DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios

Zeyu Gao[1]*, Yuxin Cui[1]*, Hao Wang[1]*, Siliang Qin[2]*
Yuanda Wang[3], Zhang Bolun[2], Chao Zhang[1]†
[1]Tsinghua University [3]Peking University
[2]Institute of Information Engineering, Chinese Academy of Sciences
{gaozy22,yx-cui24,hao-wang20}@mails.tsinghua.edu.cn
{qinsiliang,zhangbolun}@iie.ac.cn yuandawang958@stu.pku.edu.cn
chaoz@tsinghua.edu.cn

Abstract

Decompilers are fundamental tools for critical security tasks, from vulnerability discovery to malware analysis, yet their evaluation remains fragmented. Existing approaches primarily focus on syntactic correctness through synthetic micro-benchmarks or subjective human ratings, failing to address real-world requirements for semantic fidelity and analyst usability. We present DecompileBench, the first comprehensive framework that enables effective evaluation of decompilers in reverse engineering workflows through three key components: *real-world function extraction* (comprising 23,400 functions from 130 real-world programs), *runtime-aware validation*, and *automated human-centric assessment* using LLM-as-Judge to quantify the effectiveness of decompilers in reverse engineering workflows. Through a systematic comparison between six industrial-strength decompilers and six recent LLM-powered approaches, we demonstrate that LLM-based methods surpass commercial tools in code understandability despite 52.2% lower functionality correctness. These findings highlight the potential of LLM-based approaches to transform human-centric reverse engineering. We open source DecompileBench[1] to provide a framework to advance research on decompilers and assist security experts in making informed tool selections based on their specific requirements.

1 Introduction

Modern software security fundamentally depends on understanding binary code. From identifying critical vulnerabilities in the network infrastructure to analyzing sophisticated malware, security analysts rely on decompilers to bridge the semantic gap between low-level machine instructions and human-comprehensible program logic. As a crucial line of defense against evolving cyber threats, these tools must not only faithfully recover program semantics, but also generate output that facilitates rapid analysis under real-world time constraints.

Two transformative forces have reshaped the decompilation landscape. First, modern software has grown increasingly complex and aggressive compiler optimizations systematically erase high-level semantics, making decompilation increasingly harder. Second, large language models have emerged as a disruptive force in decompilation. Reverse engineers now experiment with general-purpose models like GPT-4 for reverse engineering tasks (Kwiatkowski; RevEng.AI; Ninja), while specialized models such as LLM4Decompile (Tan et al.) and MLM (AscendGrace) focus on neural decompilation. These LLM-powered approaches show promise in generating more readable code by learning high-level programming patterns.

However, these advances bring new challenges. While LLMs-powered approaches produce visually coherent code, reverse engineers question whether these decompilations preserve the semantic behaviors crucial for security analysis. This uncertainty reveals a critical gap in how we evaluate decompilers for real-world security tasks.

Crisis of Functionality Validation in Real-World Scenarios. Current decompiler evaluation approaches suffer from fundamental limitations. Most rely on artificially constructed programs (Cao et al.) (e.g., Csmith-generated code (Yang et al.)) that lack the complexity of production software. When validating these decompilers, symbolic execution methods fail due to path explosion, while unit tests (Tan et al.; Armengol-Estapé et al.) only check input-output equivalence, thus ignoring critical behaviors like global state changes, heap manipulation, and exception handling that security analysts must trace when hunting vulnerabilities.

Deficits of Automated Understandability Assessment. Current methods for evaluating decom-

*Equal contributions
†Corresponding author
[1]https://github.com/Jennieett/DecompileBench

Table 1: Decompiler Evaluation Benchmark Matrix. Symbol: ✓ (full support), ◐ (partial), ✗ (none).

Dimensions	Research Works							
	D-Helix (Zou et al.)	SAILR (Basque et al.)	LLM4Decompile (Tan et al.)	ISSTA'20 (Liu and Wang)	SEC'24 (Dramko et al.)	DecGPT (Wong et al.)	Dewolf (Enders et al.)	**Ours**
Real-world Binaries	✗	◐	◐	✗	◐	◐	◐	✓
Recompilation	◐	✗	✓	◐	✗	✓	✗	✓
Functionality Validation	◐	◐	◐	◐	◐	◐	◐	✓
Readability Metrics	✗	◐	◐	✗	◐	✗	✓	✓
Optimization Levels	✗	✓	✓	✗	✓	✓	✗	✓
Full Automation	✓	✓	◐	✓	✗	◐	◐	✓
LLM-Based Decompiler	✗	✗	✓	✗	✗	✗	✗	✓

piler output quality suffer from two key limitations. Traditional automated metrics like lines of code (LOC) and variable count (Yang et al.; Cao et al.) miss crucial semantic properties such as meaningful variable names and logical clarity. Human studies (Enders et al.; Cao et al.; Hu et al.), while providing valuable insights, lack sufficient scale across diverse compilation settings. This forces practitioners to choose between oversimplified metrics or expensive expert evaluations, neither suitable for assessing LLM-based decompilation approaches.

To bridge these existing gaps, we present `DecompileBench`, the first comprehensive evaluation framework that holistically integrates multiple key evaluation dimensions, as summarized in Table 1. Our framework advances the field through the following innovations.

Reconstructing Validation based on Runtime Consistency. We resolve the synthetic evaluation crisis through a dual-pronged approach. By establishing a production-grade assessment pipeline via the OSS-Fuzz (OSS), one of the largest continuous fuzzing frameworks for open-source software, we obtain 23,400 real-world functions from 130 actively maintained projects. We then validate decompilation correctness not through isolated unit tests or symbolic traces, but via runtime behavioral consistency during the fuzzing campaign. Our framework dynamically substitutes original functions with decompiled versions while instrumenting full-program execution paths and checking functionality correctness by evidencing through runtime behavioral across the entire binary.

Redefining Understandability with Task-Driven Metrics. To overcome the limitations of existing readability metrics, we develop a 12-dimensional assessment framework grounded in reverse engineering objectives. We employ LLM-as-a-Judge to perform scalable comparisons of decompiler outputs through security-task lenses, such as *Control Flow Clarity* and *Memory Layout Accuracy*. This automated approach, validated against expert ratings with κ=0.778 agreement, not only ranks decompilers but predicts their effectiveness for specific aspects in aiding the real-world reverse engineering tasks for reverse experts.

In summary, our main contributions are:

- **Real-World Evaluation Framework**. We develop a decompiler assessment framework to assess the decompiler from three aspects. By using real-world binary generation and runtime validation to overcome synthetic limitations, we provide reports on compiler and runtime aspects. We also use LLM-as-a-Judge for scalable decompilation assessment, replacing subjective evaluations and simplistic metrics, to conduct code quality-aspect evaluation and measure the helpfulness in human-centric reverse engineering.
- **Empirical Security Insights**. Our evaluation of 12 decompilers (6 traditional, 2 decompilation-specialized models, and 4 general-purpose LLMs) yields transformative insights. Our key findings contain Hex-Rays' 36.11% functionality correctness drop under `-O3` optimizations, LLM's significant readability improvement over Hexrays despite lower recompilation rates. These findings reshape our understanding of decompiler capabilities in security-critical contexts.
- **Open-Source Benchmark**. We release `DecompileBench` to enable systematic decompiler evaluations for future research and help analysts choose context-appropriate tools.

2 Related Works

2.1 Decompiler Evaluation

Decompilers are crucial in reverse engineering, converting binary executables into human-readable high-level source code. This task is challenging because compilers remove key information during compilation, such as variable types and control structures. Decompilers use predefined rules and heuristics to reconstruct these details (Basque et al.; Shoshitaishvili et al.), leading to significant performance variability based on their rule implementations. This variability highlights the necessity for systematic evaluation frameworks to assess decompiler quality, particularly in terms of functionality correctness and readability.

2.1.1 Functionality Correctness Evaluation

Functionality correctness evaluation aims to verify whether the decompiled code matches the original program in functionality. Prominent methods include Equivalence Modulo Inputs (EMI) testing (Liu and Wang), which compares global variable checksums between original and decompiled code executions. However, EMI relies on Csmith-generated (Yang et al.) test cases, which may oversimplify real-world complexity. To address this, Dsmith (Cao et al.) preserves intricate control and data flows by focusing on runtime-dependent variables, while D-Helix(Zou et al.) leverages real-world binaries from GitHub. Symbolic execution tools like Diff (Kim et al.), Alive (Lopes et al., b), Alive2 (Lopes et al., a), and SYMDIFF (Zou et al.) further enhance correctness verification by analyzing intermediate representations (IR) and mapping symbolic models. Despite its advantages, symbolic execution faces challenges such as path explosion, which can compromise precision.

2.1.2 Readability Assessment

Readability assessment evaluates the clarity and understandability of decompiled code, as the goal is to produce high-level representations for human understandability. Metrics such as Cyclomatic Complexity, Lines of Code, number of goto statements, and variable counts are commonly used to assess control and data flow characteristics (Cao et al.). Techniques like DREAM's pattern-independent algorithm (Basque et al.) and RevNG-C's "Control Flow Combing" aim to reduce control complexity and eliminate goto statements (Gussoni et al.). Human evaluation is also crucial. Many studies involve experienced analysts and online platforms comparing decompiler outputs based on control flow structure and code patterns (Yakdan et al.; Enders et al.; Cao et al.; Dramko et al.; Eom et al.).

2.2 Machine Learning for Decompilation

Machine learning has progressively advanced decompilation, starting with basic RNNs for binary-to-C translation (Katz et al., a) and evolving with NMT techniques that recover semantic details like variable names and types (Katz et al., b; Liang et al.; Hosseini and Dolan-Gavitt). Recent LLM-based frameworks like DecGPT (Wong et al.) and DeGPT (Lin et al.) employ hybrid methods, including static and dynamic phases, to optimize decompilation. LLM4Decompile (Tan et al.) fine-tunes LLMs to align decompiled and original code, and ReSym (Xie et al.) recovers variable semantics using LLMs combined with reasoning systems. Advanced foundation models like GPT (OpenAI) and DeepSeek (DeepSeek-AI et al.) have demonstrated an enhanced understanding of decompiled code.

2.3 LLM as a Judge

The recent advancements in large language models (LLMs) have led to the introduction of the 'LLM-as-a-judge' concept (Zheng et al.), which utilizes the capabilities of LLMs to score and rank multiple candidates (Li et al.). LLMs are capable of evaluating various aspects such as reliability (Cheng et al.), helpfulness (Lee et al., a), relevance (Lu et al.), and conducting multi-aspect assessments across diverse applications (Yu et al.). To perform pair-wise comparison and provide comprehensive feedback (Shen et al.) without positional bias, techniques such as CoT-like prompting (Zhu et al.) and output-swapping (Zheng et al.) are proposed, with tournament-based methods (Lee et al., b) further accelerating evaluations.

3 Methodology

To evaluate decompilation in real-world settings, we use the OSS-Fuzz to construct dataset. We further develop an evaluation framework that comprehensively evaluates the decompiler's performance from three aspects cared for by the end users.

3.1 Dataset Construction

The dataset construction stage, shown in Figure 1, begins with the extraction of source code from OSS-Fuzz projects (OSS). OSS-Fuzz is a platform that provides continuous fuzzing for well-known open-source software. These projects are configured and compiled with Clang's coverage sanitizer enabled, producing executable fuzzers and initial seeds as fuzzing input. By running these fuzzers

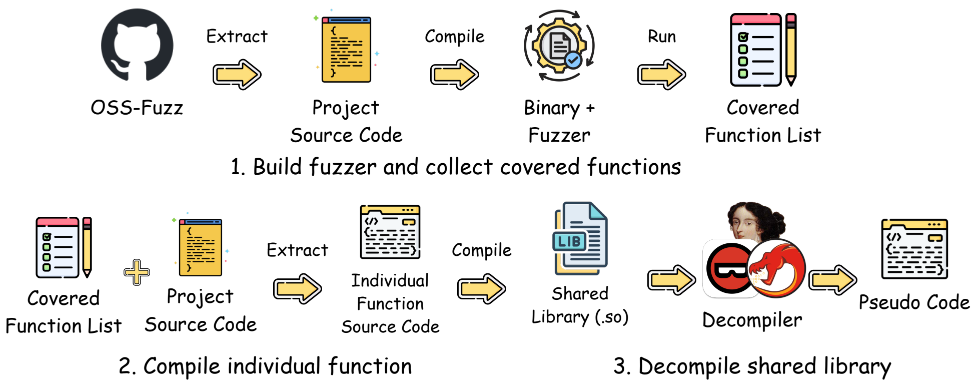
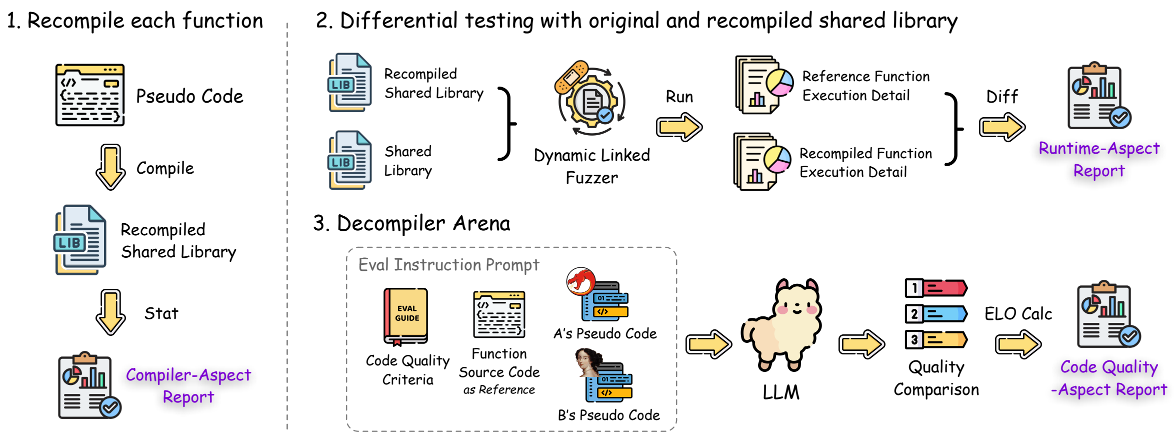


Figure 1: Overview of dataset construction process.



1. Recompile each function
Pseudo Code
Compile
Recompiled Shared Library
Stat
Compiler-Aspect Report
2. Differential testing with original and recompiled shared library
Recompiled Shared Library
Shared Library
Dynamic Linked Fuzzer
Run
Reference Function Execution Detail
Recompiled Function Execution Detail
Diff
Runtime-Aspect Report
3. Decompiler Arena
Eval Instruction Prompt
EVAL GUIDE
Code Quality Criteria
Function Source Code as Reference
A's Pseudo Code
B's Pseudo Code
LLM
Quality Comparison
ELO Calc
Code Quality -Aspect Report

Figure 2: Three-dimensional evaluation framework for decompiler assessment: Successful recompilation rate, Runtime behavior consistency, and LLM-assessed code quality.

fed with seeds as input, we utilize Clang's coverage sanitizer to identify functions covered during execution. For each covered function, we use `clang-extract` (SUS) to extract its implementation along with all dependencies, such as called function signatures, and used macro definitions. This allows us to compile individual functions into standalone binaries (shared object files, `.so`). Finally, we decompile the desired function from these binaries using multiple decompilers to obtain the decompiled code for further evaluation. We leave the detailed compile options and decompiler configurations for Section B.

3.2 Evaluation Aspect

Following the decompilation process, we evaluate the outputs across three aspects, as shown in Figure 2: the compiler aspect, the runtime aspect, and the code quality aspect. These aspects measure the re-compilation rate, functionality consistency, and the readability and practical utility of the decompiled code for reverse engineers.

3.2.1 Compiler-Aspect Report

The usability of decompiled code hinges first and foremost on its ability to meet compiler requirements, including correct language syntax and typing. To achieve this, we combine the decompiled code with the previously extracted include directives and attempt to recompile it. The result of this process is captured in a *Compiler-Aspect Report*, which quantifies the recompilation success rate. A high success rate indicates that the decompiler has preserved the essential syntactic structure and dependencies of the original code, making the decompiled output both functionally viable and practically useful.

3.2.2 Runtime-Aspect Report

While successful recompilation ensures syntactic validity, it does not guarantee that the decompiled code maintains the original program's functionality. Previous methods, such as symbolic execution (Cao et al.) and unit testing (Tan et al.), have attempted to verify decompilation accuracy but face inherent limitations. Symbolic execution struggles with path explosion in real-world programs, and unit testing focuses mainly on output equivalence, often missing complex inter-function dependencies involving global variables and multi-level pointers.

Our approach introduces an innovative side-effect consistency paradigm inspired by duck-typing (Doc) in dynamic languages. We propose that two implementations are functionally equivalent if they produce identical side effects on the program's execution environment. Practically, a decompiled function is likely correct if substituting it into the original program results in identical execution characteristics across all components. The implementation details of non-interfering function substitution are detailed in Section D.

In the evaluation, we operationalize this principle through branch coverage consistency analysis using Clang's SanitizerCoverage (LLVM) and leverage OSS-Fuzz's infrastructure in three critical phases to conduct the verification:

1. Reference Function Profiling: We substitute the target function with its original source-compiled version (a shared library), and we execute the modified program with the seed corpus collected in Section 3.1 as input, but without entering fuzzing iterations. This process collects full-program branch coverage through sanitizer instrumentation. This establishes the reference function execution profile containing ground truth coverage metrics.

2. Decompiled Code Profiling: Following the same substitution paradigm, we substitute the target function with its decompiled version while retaining the same seed corpus, then perform identical execution to generate the recompiled function execution profile.

3. Differential Analysis: After obtaining the two profiles, we compare the reference and recompiled function execution profile to check whether the decompilation preserves the control flow patterns during the whole binary execution. Here, preserving control flow patterns requires strict equivalence in execution counts of conditional statements (`if/for/while`) and boolean outcomes distribution for conditional branches (`true/false` ratios).

Moreover, this validation method can be seamlessly extended by plugging in existing instrumentation framework for finer granularity—additional comparison on stdout/stderr output, local variables (Fioraldi et al.) and operands in comparison expression (Aschermann et al.).

3.2.3 Code Quality-Aspect Report

To systematically assess decompiled code quality, we develop a dual-faceted evaluation framework focusing on *readability* (syntactic comprehension) and *helpfulness* (semantic reconstruction). We extend prior works (Dramko et al.; Cao et al.) and establish 12 fine-grained evaluation criteria spanning five readability aspects (e.g., type system consistency), five helpfulness dimensions (e.g., identifier semantics), and two hybrid criteria affecting both characteristics detailed in Table 2.

The evaluation process employs Qwen-2.5-Coder-32B (Qwen et al.) to conduct aspect-granular comparisons: For each pair of decompilers (A vs. B), the LLM i) uses the reference function code as the ground truth, ii) assesses both outputs against every criterion listed in Table 2, and iii) selects a winner for each criterion, accompanied by a detailed justification, output in a predefined JSON format. These pairwise outcomes are then used to dynamically compute Elo scores, which serve as a quantitative representation of each decompiler's code quality. To balance thoroughness and efficiency in our evaluation, we introduce a probability-based sampling strategy (detailed in Section C) that prioritizes comparisons between decompilers with similar Elo ratings. This targeted approach increases the density of comparisons among closely ranked models, enabling more precise discrimination of subtle performance differences while maintaining broad evaluation coverage.

4 Evaluation

Our evaluation encompasses six traditional decompilers, representing both commercial and open-source solutions for reverse engineering, the description and decompilation technical details are presented in Section A. For the emerging paradigm of LLM-based decompilation, we evaluate both domain-specific and general-purpose models: LLM4Decompile is tested using its official prompt template, and MLM is assessed through its public API service. The general models (GPT-4o-mini, GPT-4o, Qwen2.5-Coder-32B-Instruct, Claude-3.5-Sonnet, and DeepSeek-V3) are instructed to refine Hex-Rays outputs using task-specific guidelines and three illustrative examples as few shots. All LLM evaluations employ generation parameters with a maximum token limit of

Category	Subclass	Explanation	Example
Readability	Typecast Correctness	Redundant/incorrect type casts obscure intent	`exit((long long)"Invalid size")`
	Literal Representation	Non-idiomatic literals hinder understanding	numeric 2685 instead of string literal “\n”
	Control Flow Clarity	Complex pointer dereferencing	`for (i = a2; a1 != (*((_QWORD *)(i+64))); i = * ((_QWORD *)(i+64) ))`
	Decompiler Macros	Non-standard macros violate conventions	`LOWWORD(v5)`
	Return Behavior	Altered return expressions change logic	`return __readfsqword(0x28u)^v3`
Helpfulness	Identifier Meaning	Generic names reduce semantic value	Using `v4` instead of `buffer`
	Identifier Accuracy	Misleading variable semantics	`error_flag` vs `total_count`
	Symbolic Values	Hardcoded values reduce clarity	8 instead of `sizeof(long)`
	Function Correctness	Core functionality recovery failure	Overly too complex logic to comprehend
	Function Precision	Approximate functionality recovery	MD5 identified as SHA-256 implementation
Readability & Helpfulness	Dereference Readability	Opaque pointer arithmetic	`((_QWORD *)v5 + 8)`
	Memory Layout	Failed type inference	`(*(void (__stdcall **)(DWORD)) (*(_DWORD *)lpD3DDev_1+68))(pD3DDev_1);`

Table 2: Code Quality Aspects: 12 aspects categorized by readability, helpfulness, and both.

8,192, temperature of 0.7, and top_p of 1.0. We use Clang 18 on Ubuntu 22.04 to compile. We build our compilation service upon official OSS-Fuzz with modifications to achieve function substitution and evaluation metrics collection.

To evaluate whether LLMs can effectively capture human judgments on decompiled code quality, we employed Cohen's kappa (κ) to measure the agreement between LLM and human ratings. We describe the detail in Section E.

4.1 Compiler- and Runtime-Aspect Analysis

Our evaluation across the compiler aspect and runtime aspect are revealed through two metrics, *Recompile Success Rate* (RSR) and *Coverage Equivalence Rate* (CER), as the result shown in Table 3.

Hex-Rays is recognized as the leading traditional decompiler, achieving the highest single optimization-level RSR of 0.706 at `-O0`, and excelling in averaged metrics with an RSR of 0.583 and a CER of 0.417. Ghidra ranks second among traditional tools, establishing itself as the best open-source decompiler. In contrast, dewolf performs the worst in both metrics, as it is designed to prioritize readability for users.

Among LLM-based decompilation approaches, GPT-4o shows the highest semantic fidelity with a CER of 0.346, while GPT-4o-mini closely matches Hex-Rays' syntactic recovery capability with an RSR of 0.582. However, these LLM-enhanced results still fall short of Hex-Rays' original metrics, with RSR lower by 0.2-45.3% and CER lower by 17.2-52.2%. This is because the LLMs prioritize readability over strict compiler compatibility. General-purpose models outperform decompilation-specialized models by 69.9-120.8% in the RSR metric and by 27.3-111.6% in the CER metric. This performance gap may be due to the *temporal advantage* in their development timelines. The swift advancement of general large language models, which were released 9-12 months after the decompilation-specialized models, indicates that they can exceed the capabilities of specialized models within short technological windows.

Our experiments also reveal some key insights. First, CER generally correlates with RSR, as runtime validation inherently depends on successful recompilation. However, exceptions arise-RetDec achieves a higher RSR (0.354) than MLM (0.319) but a lower correct execution rate (CER) (0.155 vs. 0.200), indicating RetDec's prioritization of compilation over functionality. GPT-4o-mini achieves superior RSR in `-O1` to `-Os` optimizations by employing ad-hoc adaptations, such as renaming Hex-Rays' incomplete function calls from `xmlSAX2ErrMemory()` to `handle_xml_memory_error()`. This undefined function triggers a warning rather than an error during compilation into a shared library. Though this ensures compilation success, it results in runtime failures and inferior CER compared to Hex-Rays.

Second, both RSR and CER decline consistently from `-O0` to `-O3`, with `-Os` outperforming `-O3` and closely aligning with `-O2`. This trend highlights the increasing difficulty of recovering compiler-compatible code under aggressive optimization. Notably, Hex-Rays' RSR declines by 27.3% across this spectrum (from 0.706 to 0.513) and fails to achieve a CER above 50% at `-Os`. This highlights the persistent challenges of semantic recovery and emphasizes the need for significant advancements in both LLM-augmented and traditional decompilation approaches.

Decompiler	Recompile Success Rate						Coverage Equivalence Rate					
	O0	**O1**	**O2**	**O3**	**Os**	**Avg**	**O0**	**O1**	**O2**	**O3**	**Os**	**Avg**
Angr	0.309	0.232	0.190	0.181	0.191	0.221	0.187	0.153	0.124	0.116	0.118	0.140
Binja	0.274	0.246	0.229	0.215	0.224	0.238	0.167	0.153	0.137	0.129	0.138	0.145
Dewolf	0.225	0.203	0.214	0.204	0.222	0.213	0.125	0.120	0.113	0.111	0.118	0.117
Ghidra	0.524	0.421	0.395	0.377	0.353	0.413	0.374	0.294	0.256	0.241	0.228	0.278
Hex-Rays	0.706	0.573	0.558	0.513	0.565	**0.583**	0.523	0.430	0.392	0.361	0.400	**0.418**
Retdec	0.402	0.349	0.337	0.329	0.355	0.354	0.185	0.160	0.143	0.137	0.149	0.155
MLM	0.335	0.321	0.313	0.311	0.314	0.319	0.216	0.205	0.188	0.191	0.198	0.200
LLM4Decompile	0.285	0.270	0.257	0.250	0.256	0.264	0.192	0.177	0.153	0.150	0.147	0.164
Qwen2.5-Coder-32B	0.659	0.528	0.515	0.480	0.526	0.542	0.385	0.302	0.264	0.249	0.281	0.296
Deepseek-V3	0.663	0.539	0.531	0.492	0.539	0.553	0.403	0.328	0.316	0.283	0.313	0.329
GPT-4o-mini	0.658	0.591	0.559	0.531	0.572	0.582	0.296	0.269	0.231	0.210	0.244	0.254
GPT-4o	0.649	0.553	0.537	0.509	0.548	0.559	0.410	0.352	0.323	0.312	0.334	0.346
Claude-3.5-Sonnet*	0.413	0.339	0.308	0.329	0.360	0.350	0.277	0.224	0.196	0.196	0.239	0.227

Table 3: Recompile success rate and coverage equivalence rate of various decompilers across different compiler optimization levels. *Tested on a randomly sampled dataset comprising 1/5 dataset due to the high cost.

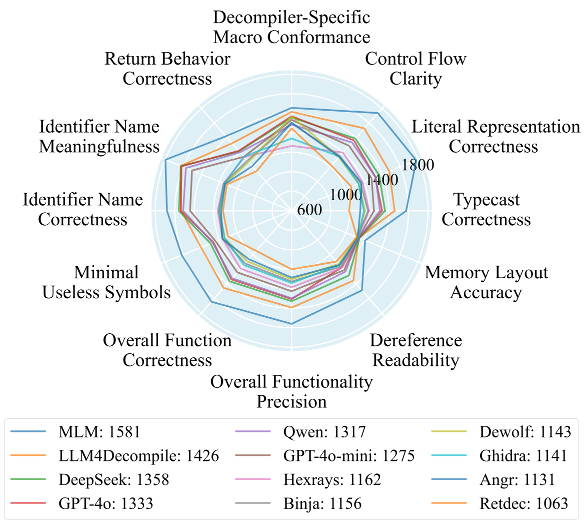


Figure 3: Comparison of code quality across twelve dimensions using Elo scores. The average Elo score across all dimensions is shown in the bottom legend. The scores are relative within each dimension, with higher scores indicating a higher win rate. Note that absolute scores across different dimensions are not directly comparable.

4.2 Code Quality Analysis

Our automated code quality assessment highlights some key insights. First, LLM-generated decompiled code consistently surpasses traditional decompilers in quality. This is true for specialized models, which benefit from training focused on readability, and general-purpose LLMs, which improve through few-shot guidance on structured code corpora. Second, MLM excels in enhancing readability, achieving an ELo score of 1581 compared to Hex-Rays' 1162, particularly excelling in *Control Flow Clarity* and *Literal Representation Correctness* (see Section F.1). However, LLMs sometimes experience hallucinations during variable inference, leading to minor semantic inaccuracies that limit improvements in *Identifier Name Correctness* and *Typecast Correctness*. Third, among traditional decompilers, Hex-Rays scores higher in readability but faces Macro Conformity issues due to the usage of nonstandard C macros, while RetDec underperforms across all metrics. Additionally, the ELo rankings for individual aspects closely match the overall rankings, indicating that decompilers rarely excel in some areas while performing poorly in others. This consistency underscores the reliability of ELo as a comprehensive measure of decompilation quality.

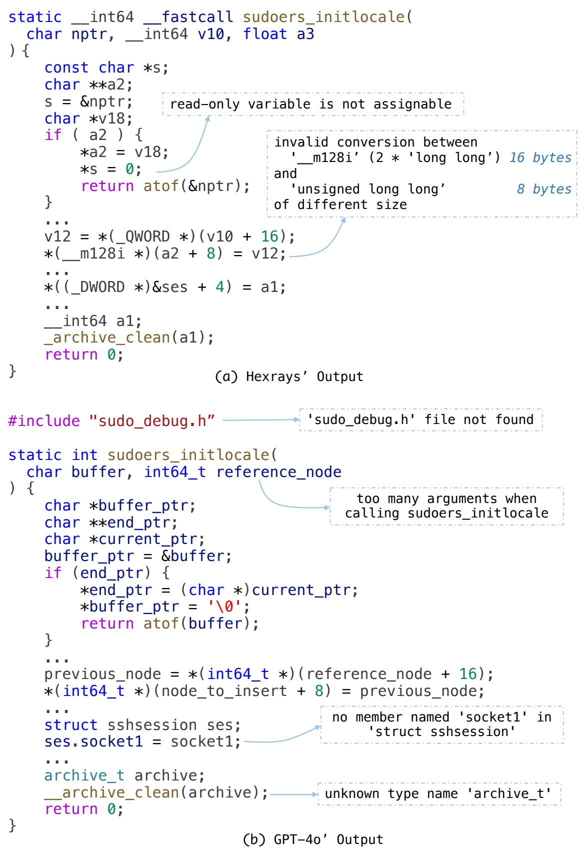
To validate the ecological validity of our LLM-based assessments, we conducted human evaluations through dual expert annotation. Two independent evaluators assessed 30 randomly selected samples, achieving an overall Cohen's kappa coefficient of 0.778. Detailed agreement statistics across quality dimensions appear in Section F.2.

4.3 From Heuristic to Neural

We evaluate traditional and LLM-based decompilers through recompilation errors, comparing their strengths and limitations to guide decompiler selection for different use cases.

4.3.1 Semantic Fidelity Tradeoffs

Our empirical study exposes fundamental divergences in semantic preservation between rule-based and neural decompilation paradigms. To systematically quantify causes of decompilation errors, we extend the error types from previous work (Cao et al.) with new types identified in our



```
static __int64 __fastcall sudoers_initlocale(
  char nptr, __int64 v10, float a3
){
    const char *s;
    char **a2;
    s = &nptr;
    char *v18;
    if ( a2 ) {
        *a2 = v18;
        *s = 0;
        return atof(&nptr);
    }
    ...
    v12 = *(_QWORD *)(v10 + 16);
    *(__m128i *)(a2 + 8) = v12;
    ...
    *((_DWORD *)&ses + 4) = a1;
    ...
    __int64 a1;
    _archive_clean(a1);
    return 0;
}
```

```
read-only variable is not assignable
```

```
invalid conversion between
  '__m128i' (2 * 'long long') 16 bytes
and
  'unsigned long long'         8 bytes
of different size
```

(a) Hexrays' Output

```
#include "sudo_debug.h"

static int sudoers_initlocale(
  char buffer, int64_t reference_node
) {
    char *buffer_ptr;
    char **end_ptr;
    char *current_ptr;
    buffer_ptr = &buffer;
    if (end_ptr) {
        *end_ptr = (char *)current_ptr;
        *buffer_ptr = '\0';
        return atof(buffer);
    }
    ...
    previous_node = *(int64_t *)(reference_node + 16);
    *(int64_t *)(node_to_insert + 8) = previous_node;
    ...
    struct sshsession ses;
    ses.socket1 = socket1;
    ...
    archive_t archive;
    __archive_clean(archive);
    return 0;
}
```

```
'sudo_debug.h' file not found
```

```
too many arguments when
calling sudoers_initlocale
```

```
no member named 'socket1' in
'struct sshsession'
```

```
unknown type name 'archive_t'
```

(b) GPT-4o' Output

Figure 4: Output of Hex-Rays and GPT-4o. The error reports highlight where one decompiler fails and the other succeeds, demonstrating their pros and cons.

compilation pipeline, culminating in 15 error types whose empirical distributions across different compilers are statistically profiled in Section G.

Traditional tools demonstrate strict adherence to low-level accuracy through deterministic pointer arithmetic (`*((_DWORD *)&ses + 4)`), yet manifest systematic limitations like type safety violations via unsafe casts (`*(__m128i *)(a2 + 8)`) or const qualification breaches during pointer dereferencing, e.g. perform `*s = 0` which is declared as readonly `const char* s`.

In contrast, LLM-based approaches (GPT-4o/MLM) achieve higher AST readability at the cost of introducing novel failure modes: Hallucinated type constructs (e.g., synthetic undefined `archive_t` replacing `__int64`); Speculative header injections (`#include "sudo_debug.h"` which is not in the context); Critical parameter omission in function ABIs, breaking function invocations by removing seemingly unused parameters (e.g., `float a3`), even when such parameters are critical to runtime compatibility.

The neural paradigm particularly struggles with pointer arithmetic resolution, as evidenced in Figure 4 where GPT-4o generates invalid struct member access (`ses.socket1`) versus Hex-Rays' bit-precise implementation. Hybrid approaches show promise in bridging this gap, as demonstrated by LLM-corrected type casts (`*(int64_t *)(node_to_insert + 8)`) and relaxed type constraints (e.g., redefining `*buffer_ptr` as mutable `char`), surpassing traditional tools' output validity.

4.3.2 Scenario-Driven Tool Choosen

These findings reveal a trade-off between readability and correctness, suggesting the need to combine LLMs' contextual flexibility with traditional tools' rigorous type-checking in practical applications.

For reliability-critical scenarios like performance analysis and debugging, established decompilers such as Hex-Rays and Ghidra remain preferable due to their semantically accurate and dependable outputs, even if they are sometimes less user-friendly. Conversely, in reverse engineering where quick comprehension is crucial, such as malware detection, LLM-based decompilers are preferred. Especially those models fine-tuned for clarity, like MLM, offer enhanced readability that significantly aids analysis. Ultimately, the choice of decompilation technique should be guided by the specific analytical objectives, balancing stringent accuracy with human interpretability.

5 Conclusion

We present `DecompileBench`, a comprehensive decompiler evaluation framework that addresses real-world assessment challenges through three key innovations: production-grade datasets from OSS-Fuzz, runtime behavior validation, and LLM-powered code quality analysis. Our evaluation of 12 decompilers highlights critical trade-offs: traditional tools such as Hex-Rays achieve a 58.3% average recompilation success rate, whereas LLM-based approaches like MLM excel in code quality, offering superior control flow clarity and more meaningful identifier naming.

This work establishes new standards for decompiler evaluation in security-critical scenarios and highlights the need for hybrid approaches combining neural-based decompilers with rule-based approaches. Our framework enables informed tool selection for reverse engineers while guiding future research toward balancing reliability with human-centric readability for reverse engineering.

6 Limitation

6.1 Error Localization

Our analysis evaluates recompilation failures as semantic fidelity indicators for decompilation quality. While runtime differential coverage metrics detect functional discrepancies between original and decompiled outputs, opaque dependencies in shared libraries hinder precise error localization.

6.2 Root-Cause Diagnostic

Our methodology focuses on *where* decompilers underperform, but the complexity of decompilation obscures the *why* behind these failures, preventing root-cause analysis. This underscores the need for further exploration, such as differential testing of heuristics, to improve the internal mechanisms of open-source decompilers, as demonstrated by studies like (Zou et al.).

6.3 LLM-Augmented Reprocessing

To address compilation errors with LLMs, we did not integrate Clang-extracted dependencies (e.g., typedefs, structs, macros, includes) into LLM inputs due to resource constraints. This integration could enable joint inference of type definitions and resolution of external calls. Moreover, feeding error information to LLMs and iteratively recompiling their outputs could enhance decompilers. However, due to time constraints, this remains a task for future work.

6.4 Architecture-Specific Evaluation

The evaluation is restricted to x86-64 architectures, despite platform-dependent variations in shared library behavior and compiler optimizations. For example, decompiled functions relying on x86-64-specific features like register usage or memory alignment may fail on ARM architectures.

6.5 Limited Human Validation

While achieving expert-LLM agreement (κ=0.778) on code quality metrics, the validation is limited to a small set of cases (n=30) evaluated by only two annotators. This narrow scope risks overlooking diverse human perspectives and potential ‘reward hacking’, where models optimize for evaluation patterns rather than genuine quality improvements. Expanding to diverse architectures and a larger annotator pool would enhance the generalizability and robustness of our findings.

References

OSS-Fuzz.

SUSE/clang-extract.

Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley De Souza Magalhães, and Michael F. P. O'Boyle. ExeBench: An ML-scale dataset of executable C functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 50–59. ACM.

AscendGrace. Machine Language Model: AI Empowerment, Gaining Insight into the Binary World.

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings 2019 Network and Distributed System Security Symposium*. Internet Society.

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In *33st USENIX Security Symposium (USENIX Security 24)*.

Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the Effectiveness of Decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 491–502. ACM.

Qinyuan Cheng, Tianxiang Sun, Wenwei Zhang, Siyin Wang, Xiangyang Liu, Mozhi Zhang, Junliang He, Mianqiu Huang, Zhangyue Yin, Kai Chen, and Xipeng Qiu. Evaluating Hallucinations in Chinese Large Language Models. *Preprint*, arXiv:2310.03368.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. DeepSeek-V3 Technical Report. *Preprint*, arXiv:2412.19437.

Python Doc. Glossary.

Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. A Taxonomy of C Decompiler Fidelity Issues. In *33rd Usenix Security Symposium*.

Steffen Enders, Eva-Maria C. Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. Dewolf: Improving Decompilation by leveraging User Surveys. In *Proceedings 2023 Workshop on Binary Analysis Research*.

Haeun Eom, Dohee Kim, Sori Lim, Hyungjoon Koo, and Sungjae Hwang. R2I: A Relative Readability Metric for Decompiled Code. 1:18:383–18:405.

Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The Use of Likely Invariants as Feedback for Fuzzers. pages 2829–2846.

Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A Comb for Decompiled C Code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, pages 637–651. Association for Computing Machinery.

Iman Hosseini and Brendan Dolan-Gavitt. Beyond the C: Retargetable Decompilation using Neural Machine Translation. In *Proceedings 2022 Workshop on Binary Analysis Research*.

Peiwei Hu, Ruigang Liang, and Kai Chen. DeGPT: Optimizing Decompiler Output with LLM. In *Proceedings 2024 Network and Distributed System Security Symposium*. Internet Society.

Deborah S. Katz, Jason Ruchti, and Eric Schulte. a. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE.

Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. b. Towards Neural Decompilation. *Preprint*, arXiv:1905.08325.

Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364. IEEE.

Ivan Kwiatkowski. JusticeRage/Gepetto.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, and Sushant Prakash. a. RLAIF: Scaling Reinforcement Learning from Human Feedback with AI Feedback. *Preprint*, arXiv:2309.00267.

Sangkyu Lee, Sungdong Kim, Ashkan Yousefpour, Minjoon Seo, Kang Min Yoo, and Youngjae Yu. b. Aligning Large Language Models by On-Policy Self-Judgment. *Preprint*, arXiv:2402.11253.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. From Generation to Judgment: Opportunities and Challenges of LLM-as-a-judge. *Preprint*, arXiv:2411.16594.

Ruigang Liang, Ying Cao, Peiwei Hu, Jinwen He, and Kai Chen. Semantics-Recovering Decompilation through Neural Machine Translation. *Preprint*, arXiv:2112.15491.

Chi-Heng Lin, Shangqian Gao, James Seale Smith, Abhishek Patel, Shikhar Tuli, Yilin Shen, Hongxia Jin, and Yen-Chang Hsu. MoDeGPT: Modular Decomposition for Large Language Model Compression. *Preprint*, arXiv:2408.09632.

Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487. ACM.

LLVM. SanitizerCoverage — Clang documentation.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. a. Alive2: Bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 65–79. Association for Computing Machinery.

Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. b. Provably correct peephole optimizations with alive. 50(6):22–32.

Junyi Lu, Xiaojia Li, Zihan Hua, Lei Yu, Shiqi Cheng, Li Yang, Fengjun Zhang, and Chun Zuo. DeepCRCEval: Revisiting the Evaluation of Code Review Comment Generation. *Preprint*, arXiv:2412.18291.

Binary Ninja. Binary Ninja Sidekick: Your AI Reverse Engineering Assistant.

OpenAI. GPT-4 Technical Report.

Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report. *Preprint*, arXiv:2412.15115.

RevEng.AI. RevEng.AI: Reverse Engineering meets Artificial Intelligence.

Yanxin Shen, Lun Wang, Chuanqi Shi, Shaoshuai Du, Yiyi Tao, Yixian Shen, and Hang Zhang. Comparative Analysis of Listwise Reranking with Large Language Models in Limited-Resource Language Contexts. *Preprint*, arXiv:2412.20061.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling Binary Code with Large Language Models. *Preprint*, arXiv:2403.05286.

Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining Decompiled C Code with Large Language Models. *Preprint*, arXiv:2310.06530.

Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries.

Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. 46(6):283–294.

Zhuohao Yu, Chang Gao, Wenjin Yao, Yidong Wang, Wei Ye, Jindong Wang, Xing Xie, Yue Zhang, and Shikun Zhang. KIEval: A Knowledge-grounded Interactive Evaluation Framework for Large Language Models. *Preprint*, arXiv:2402.15043.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. *Preprint*, arXiv:2306.05685.

Banghua Zhu, Evan Frick, Tianhao Wu, Hanlin Zhu, Karthik Ganesan, Wei-Lin Chiang, Jian Zhang, and Jiantao Jiao. Starling-7B: Improving Helpfulness and Harmlessness with RLAIF.

Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, and Antonio Bianchi. D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *The Proceedings of the 33rd USENIX Security Symposium*.

A Decompiler Details

This appendix provides description and technical details about each decompiler in our paper:

- **Angr**: Binary analysis framework utilizing proprietary AIL (angr Intermediate Language) combining symbolic execution with Value-Set Analysis (VSA). Executes decompilation through `angr.Project.Decompiler` class via direct invocation on target functions. Supports x86, ARM, and MIPS architectures.
- **Binary Ninja**: Multi-stage decompilation platform with LLIL/MLIL/HLIL intermediate representations. Creates decompilation views via `LinearViewObject.single_function_language_representation(func, settings)`, constructing final output through iterative cursor object traversal.
- **Dewolf**: Binary Ninja-based decompiler optimized for human-readable C code. Initializes decompilation process using `DecompilerPipeline.from_strings()` that ingests control flow graph (CFG) and abstract syntax tree (AST) inputs, subsequently executing the pipeline for code generation.
- **Ghidra**: Enterprise-grade system using P-Code intermediate language. Invokes decompilation through `FlatDecompilerAPI(flat_api)` interface followed by explicit `decomp_api.decompile(function)` calls in the API execution chain.
- **Hex-Rays**: Industry-standard decompiler implementing `ida_hexrays.decompile(func)` for single-step decompilation within IDA Pro's interactive environment. Features production-grade quality through mature pattern recognition.
- **RetDec**: LLVM-based decompilation framework using multi-pass optimization. Executes standalone `retdec-decompiler` executable (located in installation directory) directly on target binaries to initiate the decompilation process.

B Compilation and Decompilation Detail

When compiling fuzzer and executable, we append `-Wl,-export-dynamic` to compile commands, enabling shared library to resolve symbols in the executable in function substitution 3.2.2. When compiling target functions into shared libraries, we add `-fno-inline` to prevent function inlining that would obstruct individual function extraction from the binary. After obtaining the decompiled code, we identify common error patterns for each decompiler and conduct general post-processing corrections. For example, in Binary Ninja, we remove patterns such as `@ zmm0`[2] and `__pure`[3] that could absolutely cause compilabtion errors.

C Sampling Strategy in LLM-as-a-Judge

Initially, for each binary in our dataset, we randomly select the output of one decompiler. Subsequently, we calculate the probability of selecting another decompiler (model b) for comparison, based on their ELo scores. Let R_a and R_b represent the ELo scores of models a and b, respectively. The selection probability $P(b)$ for model b is determined as follows:

$$P(b) = \frac{1}{1 + \frac{|R_a - R_b|}{\min\limits_i |R_a - R_i| + \epsilon}} \tag{1}$$

where ϵ is a small constant (1e-6) to avoid division by zero, and R_i represents the rating of any model i in the candidate pool. The final normalized probability is:

$$P_{norm}(b) = \frac{P(b)}{\sum\limits_j P(j)} \tag{2}$$

This approach effectively prioritizes the evaluation of more comparable decompiler outputs while still maintaining comprehensive coverage. Formally, the selection probability $P(d_i)$ for decompiler d_i is proportional to the proximity of its ELo score to the reference.

D Function Substitution Implementation Details

In this section, we elaborate on the technical implementation of our function substitution mechanism, i.e. substituting the decompiled function into the original program without influence any other function (including the virtual address), comprising two critical phases: compile-time preparation and runtime redirection.

[2]use `zmm0` register to store the function argument

[3]Binary Ninja-specific function attribution

```c
//headers
#define bool_test bool
/** clang-extract: from test.h  */
bool_test test(int input);
// target function
static bool_test function_test(int a)
{
    result = test(a);
    return result;
}
__attribute__((constructor)) void initializer()
{
    void **address = (void **)syscall(
        MMAP, (void*)0xbabe0000, 4096,
        PROT_READ | PROT_WRITE,
       MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED,
       -1, 0
    );
    if (address == MAP_FAILED)
        syscall(EXIT, 1);
    *(void **)(0xbabe0000) = function_test;
}
__attribute__((destructor)) void finalizer()
{
    syscall(MUNMAP, (void *)0xbabe0000, 4096);
}
```

Figure 6: Generating a complete compilable C test file: adding necessary headers initializer and finalizer around the target function 'function_test'.

D.1 Compile-Time Preparation

To establish control over the executable's dynamic linking process, we modify the fuzzer compilation commands to force the compilation links against a specially crafted dummy shared library (dummy.so). This shared library contains a mandatory entry point for OSS-Fuzz's execution framework, LLVMFuzzerTestOneInput. This ensures our dummy library gets loaded through dynamic linking. The dummy shared library contains no operational logic, serving only as a placeholder to inject our instrumentation later. The compilation command modification preserves symbol visibility for subsequent dynamic function calling by adding -Wl,-export-dynamic in the compilabtion commands.

D.2 Runtime Redirection

During actual execution, we replace the dummy implementation with our instrumented target function through the following coordinated steps.

D.2.1 Function Prologue Patching

We locate the function to be substituted in the executable and we replace the function prologue, the initial set of instructions in an function, with custom machine code shown in 5 that redirects execution to an address hold by a memory-mapped fixed location, 0xbabe0000.

```
xor rax, rax;
mov eax, 0xbabe0000;
mov rax, [rax];
jmp rax;
```

Figure 5: Function prologue patching to redirect execution to a fixed memory-mapped location.

D.2.2 Address Binding

We insert two auxiliary functions, an initializer and a finalizer, into the shared object. The initializer function is called when the shared object is loaded, and the finalizer is called when the shared object is unloaded. The initializer function stores the address of the target into a fixed global address (0xbabe0000) to facilitate later runtime redirection. And the finalizer is responsible for cleaning up. We show the code snippet in Figure 6.

D.2.3 Resolving External Function Calls

A notable characteristic of real-world programs is that functions rarely operate in isolation; most invoke other functions within the project. When testing the compilation success rate and functionality correctness of individual decompiled functions, we must include declarations of all external functions as the context. Ideally, during execution, these external functions should use their original implementations in the binary.

When we take a function that has been decompiled and compiled back into a shared object (.so), and then try to use this .so alongside the original binary, we encounter a specific issue. If the function in our .so calls another function, say b, which exists and is implemented only within the original binary, and that binary has not made b available for dynamic linking (meaning b is not listed in the binary's dynamic symbol table, dynsym), then the .so file will contain a reference to b but no implementation.

During execution, when the code in the .so attempts to call b, the dynamic linker tries to find b's implementation. Because b is not visible in the binary's dynamic symbols, the linker fails to resolve the symbol, resulting in a runtime error, typically a "symbol lookup error: ‘b’". This problem arises because the binary’s internal functions are not fully exposed for dynamic linking, and since we are working only with the compiled binary, we cannot alter its compilation settings to change this.

To resolve this, we need to ensure that calls from the `.so` to functions implemented in the binary are correctly directed. This process effectively involves manually performing some of the relocation tasks that the dynamic linker (ld) would typically handle. Before execution, we collect specific addresses for relevant functions in both the binary and the `.so`.

- For functions implemented within the binary: We determine the address where the function’s actual code resides. This is the address associated with the function’s symbol.
- For calls made from the `.so` to external functions (like those in the binary): We identify the address within the `.so`’s `.got.plt` section that corresponds to the symbol of the function being called. This `.got.plt` entry is the point within the `.so` that needs to be adjusted to correctly call the external function.

By obtaining these addresses, along with the runtime base addresses of the binary and the `.so`, we can calculate the final runtime addresses and redirect the external calls within the `.so` to point directly to the implementations within the binary.

E Cohen’s Kappa

Cohen’s kappa measures the agreement between two raters classifying N items into C mutually exclusive categories, defined as:

$$\kappa \equiv \frac{p_o - p_e}{1 - p_e} = 1 - \frac{1 - p_o}{1 - p_e} \tag{3}$$

where p_o represents the relative observed agreement between raters, and p_e represents the hypothetical probability of chance agreement. For k categories and N observations, with n_{ki} representing the number of times rater i predicted category k, p_e is calculated as:

$$p_e = \frac{1}{N^2} \sum_{k=1}^{C} n_{k1} n_{k2} \tag{4}$$

This calculation of p_e is derived from the assumption that ratings are independent, where the probability of each rater classifying an item as category k is estimated by the proportion of items they assigned to that category: $\widehat{p_{k1}} = \frac{n_{k1}}{N}$ (and similarly for rater 2).

$$p_o = \frac{1}{N} \sum_{k=1}^{C} n_{kk} \tag{5}$$

The observed agreement p_o is calculated using n_{kk}, which counts items assignegned to category k by both raters. The coefficient ranges from -1 to 1, where $\kappa = 1$ indicates perfect agreement, $\kappa = 0$ suggests agreement no better than chance, and negative values indicate systematic disagreement.

F Code Quality

F.1 Case Study

As illustrated in Figure 7, MLM markedly improves control flow clarity by accurately recovering `switch` statements and assigning semantically meaningful labels to conditional branches. In addition, it enhances code readability through sophisticated pointer dereference resolution, effectively transforming low-level constructs such as `*((_DWORD *)ptr + 2) = a1` into more human-readable forms like `leak->type = type`, a feature that is also partially achieved by DeepSeek.

F.2 Agreement Analysis

Table 4 demonstrates significant agreement between LLM and human evaluators across these aspects: Non-idiomatic Literal Representation ($\kappa = 0.905$), Non-idiomatic Dereferencing ($\kappa = 0.866$), and Expanded Symbols ($\kappa = 0.917$). However, notable discrepancies emerged in other areas, primarily due to ambiguous evaluation criteria:

- **Meaningless Identifiers** ($\kappa = 0.530$): The LLM occasionally provides judgments contrary to the established rules, penalizing meaningful names by labeling code with genuinely meaningful variable names as "loser." This discrepancy likely stems from misapplied evaluation criteria.
- **Incorrect Identifiers** ($\kappa = 0.656$): The LLM tends to classify variable names with slight semantic deviations as incorrect, while overlooking entirely meaningless names. In contrast, human annotators focus on significant semantic deviations from the source code variable names, considering only misleading identifiers as incorrect. This divergence highlights the LLM’s limited capability in assessing the misleading nature of identifiers.

```
switch (a1) {
  case 1u:
    ptr[2] = a2;
    goto LABEL_14;
  case 2u:
    ptr[2] = a2;
    goto LABEL_14;
  case 3u:
    ptr[2] = a2;
    goto LABEL_14;
  case 4u:
    ptr[2] = a2;
    goto LABEL_14;
  case 5u:
    ptr[2] = a2;
    goto LABEL_14;
  case 6u:
    ptr[2] = a2;
  LABEL_14:
    *((_DWORD *)ptr + 2) = a1;
    *ptr = parser_leak_list;
    parser_leak_list = (__int64)ptr;
    result = 1LL;
    break;
  default:
    free(ptr);
    sudo_warnx_nodebug_v1("unexpected
leak type %d", a1);
    result = 0LL;
    break;
}
```

(a) Hexrays's decompiled code

```
switch (type) {
  case PARSER_LEAK_ALIAS:
    leak->ptr = ptr;
    break;
  case PARSER_LEAK_ALIAS_LIST:
    leak->ptr = ptr;
    break;
  case PARSER_LEAK_DEFAULTS:
    leak->ptr = ptr;
    break;
  case PARSER_LEAK_DEFAULTS_LIST:
    leak->ptr = ptr;
    break;
  case PARSER_LEAK_HOSTALIAS:
    leak->ptr = ptr;
    break;
  case PARSER_LEAK_HOSTALIAS_LIST:
    leak->ptr = ptr;
    break;
  case PARSER_LEAK_RUNAS:
    leak->ptr = ptr;
    break;
  default:
    free(leak);
    sudo_warnx_nodebug("unexpected
leak type %d", type);
    return false;
}
leak->type = type;
leak->next = parser_leak_list;
parser_leak_list = leak;
return true;
```

(b) MLM's decompiled code

```
switch (leak_type) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
  case 6:
    new_entry[2] = leak_data;
    ((unsigned int *)new_entry)[2] = leak_type;
    new_entry[0] = parser_leak_list;
    parser_leak_list = new_entry;
    return 1;
  default:
    free(new_entry);
    sudo_warnx_nodebug_v1("unexpected leak type %d", leak_type);
    return 0; }
```

(c) GPT-4o's decompiled code

```
switch (leak_type) {
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
  case 6:
    entry->data = leak_data;
    entry->type = leak_type;
    entry->next = parser_leak_list;
    parser_leak_list = entry;
    return 1;
  default:
    free(entry);
    sudo_warnx_nodebug_v1("unexpected leak type %d", leak_type);
    return 0; }
```

(d) DeepSeek's decompiled code

Figure 7: Cases related to the code quality of Hexrays, MLM, GPT-4o, and DeepSeek.

Issue Type	κ	Complete agreement
Typecast Correctness	0.868	0.933
Literal Representation Correctness	0.905	0.97
Control Flow Clarity	0.826	0.933
Decompiler-Specific Macros	0.704	0.87
Return Behavior Correctness	0.776	0.00900
Identifier Name Meaningfulness	0.530	0.77
Identifier Name Correctness	0.656	0.833
Minimal Useless Symbols	0.917	0.97
Overall Function Correctness	0.809	0.00900
Overall Functionality Precision	0.769	0.00900
Dereference Readability	0.866	0.999
Memory Layout Accuracy	0.727	0.87

Table 4: LLM-Rater agreement results.

- **Memory Layout Abuse** ($\kappa = 0.727$): The LLM adopts a stricter evaluation approach compared to the more lenient human assessments. As a result, code deemed acceptable by humans may be flagged by the LLM as exhibiting Memory Layout Abuse.

G Error Analysis

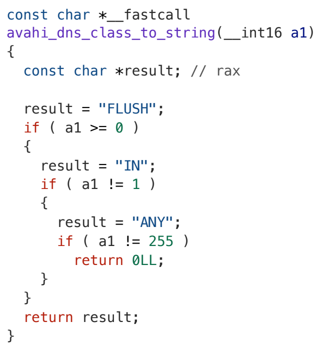
Our comprehensive evaluation scrutinizes compile-stage errors across decompilers. While previous studies (Zou et al.; Eom et al.; Cao et al.) identified multiple fundamental error types based on errors observed during the recompile phase of traditional decompilers, our analysis of LLM-based decompilers necessitated a revised taxonomy that accounts for novel error categories and their distribution. Building upon prior work, we introduce additional error types—such as *control flow issues*, *memory issues*, *file and resource issues*, and *type conversion/compatibility* errors—thereby establishing a phase-aware framework that delineates 15 distinct error types. Besides, category A focuses on Assembly Issues, dealing with errors related to interpreting and converting assembly code. Category B addresses Variable and Memory Issues, including problems with variable declarations, memory management, and type conversions. Category C highlights Function and Control Flow Issues, which involve errors in function resolution, control flow, and handling of user-defined types. Category D deals with Syntax and Macro Issues, which include errors related to syntax, expressions, and preprocessor macros during the translation process. This refined classification provides a comprehensive basis for understanding and addressing the challenges inherent in the decompilation process.

Traditional decompilers exhibit a mix of strengths and weaknesses across error categories. Hex-Rays excels in precise control flow recovery with zero *Control Flow Issues (X)*, in Figure 8, ensuring that the output accurately reflects the original logic without introducing execution errors. While Angr overuses unstructured jumps (`goto`), resulting in 255 errors. Binja, though generally effective in structuring control flow, occasionally merges unrelated branches, leading to ambiguity.

Error Category	Error	Traditional Decompiler						Decompilation-specialized LLMs		General LLM			
		Angr	Binja	Dewolf	Ghidra	Hexrays	Retdec	MLM	LLM4Decompile	Qwen2.5	Deepseek-V3	GPT-4o-mini	GPT-4o
A	Assembly Issues (**I**)	335	118	331	12	27	29	173	204	430	165	167	154
	File and Resource Issues (**II**)	-	-	-	-	-	-	-	-	17	-	32	33
B	Variable Declaration/Naming (**III**)	822	886	863	4414	2237	6508	3289	2325	2487	1956	2854	2538
	Memory Issues (**IV**)	43	23	-	-	12	-	6	10	29	13	14	9
	Type Conversion/Compatibility (**V**)	1079	2006	1229	486	875	1066	126	77	622	876	663	655
	Initialization Issues (**VI**)	-	-	219	-	-	-	71	122	103	89	83	87
C	Function Declaration/Invocation (**VII**)	4863	2626	3008	4417	3470	3483	1457	1655	4254	3862	3411	3220
	User-Defined Type Issues (**VIII**)	665	475	336	279	209	321	2339	3038	483	824	347	294
	Dependency/Redefinition (**IX**)	2938	1842	2103	1492	1970	1761	188	160	2138	1837	2081	2056
	Control Flow Issues (**X**)	255	607	56	76	-	304	44	85	45	28	32	21
D	Target Platform/Config (**XI**)	-	12	16	22	10	27	-	-	12	14	11	12
	Type Definition/Resolution (**XII**)	1200	1912	2108	527	16	96	7705	10066	692	890	528	488
	Expression/Operator (**XIII**)	5601	7264	9898	800	267	348	472	563	827	635	787	536
	Syntax Errors (**XIV**)	2484	432	1233	855	103	238	109	314	171	36	91	225
	Macro/Preprocessor (**XV**)	7	4	2	5	6	8	1	7	7	-	1	6

Table 5: Comparison of Error Types Across Traditional Decompilers, LLM Decompilers, and General LLMs

In *Expression/Operator Handling (XIII)*, Hex-Rays (267) and RetDec (348) outperform others by accurately recovering complex bitwise operations, such as sign extension and masking. In contrast, Angr (5601) misinterprets bitwise operations as arithmetic, and Binja(7264) struggles to unify shifts and masks. DeWolf (9,898) and some LLM decompilers exhibit high error rates due to fragmented expressions and logical oversimplifications.

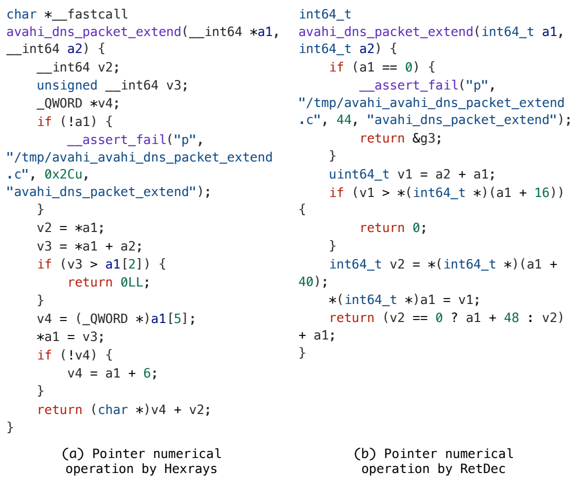


```
const char *__fastcall
avahi_dns_class_to_string(__int16 a1)
{
  const char *result; // rax

  result = "FLUSH";
  if ( a1 >= 0 )
  {
    result = "IN";
    if ( a1 != 1 )
    {
      result = "ANY";
      if ( a1 != 255 )
        return 0LL;
    }
  }
  return result;
}
```

Figure 8: Cases related to the control flow recovery in Hexrays.

HexRays and Retdec excel in handling complex bitwise operations and accurately recovering high-level semantics, such as sign extension, shifting, and masking. Specific examples in the provided implementations(Figure 9 demonstrate their advantages. In the HexRays output, line 11 combines pointer arithmetic and bounds-checking effectively (`v3 = *a1 + a2`), ensuring correctness while preserving clarity. Line 20 integrates fallback pointer logic with precise pointer arithmetic (`return (char *)v4 + v2`), highlighting HexRays' ability to produce accurate and readable code. Similarly, the Retdec implementation shines in line 34 by seamlessly handling conditional logic for fallback pointers (`return (v2 == 0 ? a1 + 48 : v2) + a1`), which maintains high semantic fidelity while combining conditional expressions. These strengths enable both tools to provide reliable, post-process-ready outputs with minimal need for manual corrections, outperforming traditional decompilers in accuracy and readability.



```
char *__fastcall
avahi_dns_packet_extend(__int64 *a1,
__int64 a2) {
    __int64 v2;
    unsigned __int64 v3;
    _QWORD *v4;
    if (!a1) {
        __assert_fail("p",
"/tmp/avahi_avahi_dns_packet_extend
.c", 0x2Cu,
"avahi_dns_packet_extend");
    }
    v2 = *a1;
    v3 = *a1 + a2;
    if (v3 > a1[2]) {
        return 0LL;
    }
    v4 = (_QWORD *)a1[5];
    *a1 = v3;
    if (!v4) {
        v4 = a1 + 6;
    }
    return (char *)v4 + v2;
}
```

(a) Pointer numerical operation by Hexrays

```
int64_t
avahi_dns_packet_extend(int64_t a1,
int64_t a2) {
    if (a1 == 0) {
        __assert_fail("p",
"/tmp/avahi_avahi_dns_packet_extend
.c", 44, "avahi_dns_packet_extend");
        return &g3;
    }
    uint64_t v1 = a2 + a1;
    if (v1 > *(int64_t *)(a1 + 16))
{
        return 0;
    }
    int64_t v2 = *(int64_t *)(a1 +
40);
    *(int64_t *)a1 = v1;
    return (v2 == 0 ? a1 + 48 : v2)
+ a1;
}
```

(b) Pointer numerical operation by RetDec

Figure 9: Cases related to the complex operator in HexRays and RetDec.

In contrast, our error analysis shows that Angr misinterprets bitwise operations as arithmetic and often omits crucial details like sign extension. Binja, while capable of handling basic bitwise operations, struggles to combine shifts and masks into cohesive expressions, leading to less accurate recovery. LLM-based approaches frequently fail to interpret complex operations correctly, introducing logical errors or oversimplifi-cations, particularly in edge cases. DeWolf often fragments expressions, reducing readability and risking incorrect optimization when recompiled. Additionally, its simplification of bounds-checking expressions can compromise runtime safety and correctness in edge cases.

Decompilation-specialized models such as MLM and LLM4Decompile significantly outperform other methods in resolving *Dependency/Redefinition Issues (IX)*, with error counts of 188 and 160, respectively. In contrast, traditional decompilers, despite incorporating domain-specific fixes (e.g., handling includes, typedefs, and defines), inadvertently introduce redefinition conflicts with Clang-extracted include statements, as evidenced by Angr's 2938 errors. Similarly, general LLM-based decompilers exacerbate these issues by inserting custom macros and headers (e.g., Qwen: 2138 errors). However, specialized models exhibit pronounced weaknesses in *Type Definition/Resolution (XII)* and *User-Defined Type Issues (VIII)*, with error counts starkly exceeding those of traditional tools like Hex-Rays and RetDec and even lagging behind general LLMs. This stems from their prioritization of readability and dependency simplification over precise type inference, often replacing complex pointer dereferences with fabricated types.

General LLMs, while robust in type-related tasks due to pretraining on diverse code patterns, struggle with *File and Resource Issues (II)*, *Function Declaration/Invocation (VII)*, and *Dependency/Redefinition Issues (IX)*, frequently inserting nonexistent headers, altering function parameters, introducing external dependencies or rewriting external call functions. Across all methods, type, variable, and function-related errors dominate the failure modes, underscoring the persistent challenges in balancing syntactic correctness with semantic fidelity.